Form is Function:

Addressable Computation in Evolving State Machines with Computable Connections

Daniel Scott Matthews

Theoretical Computer Science · Manuscript Draft · 2026

Abstract

In traditional computational models, *form* (structure, encoding, or description) and *function* (behavior or transition rules) are treated as distinct layers. This paper introduces **meta-machines**, a higher-order model in which a state machine's current configuration encodes a complete, computable description of its successor machine. Through **addressable computation**—stable references realized as Gödel numbers, pointers, or cryptographic content hashes—each state deterministically derives and references the next machine, making form literally function: the encoded structure *is* the functional specification of the successor.

Building on Gurevich's Abstract State Machines (ASMs), self-modifying finite automata (SMFAs) [16], content-addressable paradigms (IPFS [1], IPFS-FAN [5]), Dhaliwal's addressability thesis [6], and reflective systems [19], we formalize meta-machines with a **typed transfer function** enforcing schema compatibility across machine generations. We prove Turing completeness, establish that the meta-halting problem is Σ_1^0-complete, and derive a meta-fixed-point theorem via Kleene's second recursion theorem [15]. We define a **strict hierarchy of computable-connection classes** with explicit diagonalization separation sketches, and provide categorical coalgebraic and higher-order rewriting embeddings that clarify equivalences to known models.

The central claim—that the addressability constraint yields unique structural properties (verifiable computation lineage, deterministic replay, composable fork semantics) absent in classical reflection or unstructured self-modification—is defended against prior art ranging from SMFAs and certified self-modifying code [2] through ex-machines [7] and the Darwin Gödel Machine [22]. Applications span decentralized Web3 computation, reflective AI architectures, fault-tolerant dynamic reconfiguration, and quantum/neuromorphic substrates.

Keywords: addressable computation, meta-machines, computable-connection hierarchy, Abstract State Machines, self-modifying finite automata, content-addressable storage, IPFS-FAN, Kleene recursion theorem, higher-order rewriting, coalgebra, distributed systems.

1. Introduction: Beyond Static Form and Function

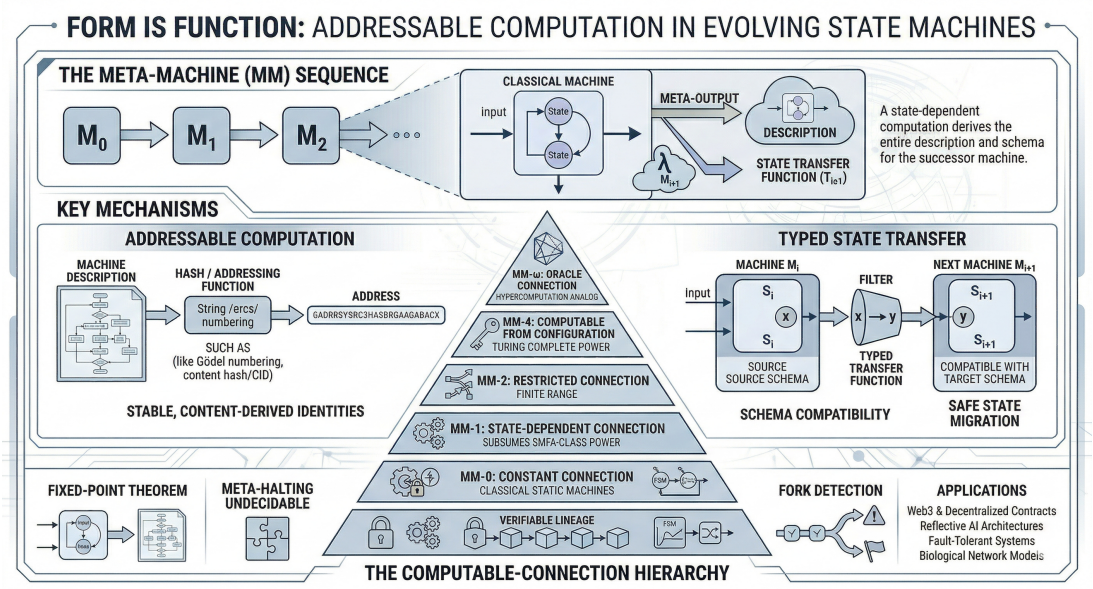

Figure 1: Meta-Machine Evolution and the Computable-Connection Hierarchy.

The architectural maxim "form follows function" (Louis Sullivan, 1896) and its functional-structuralist inversion "form is function" highlight a deep interplay between structure and behavior. In computation, the von Neumann stored-program concept blurred instructions and data, yet retained a *fixed interpreter*: the machine description itself does not evolve as a first-class citizen of the computation.

We propose a model in which this separation dissolves entirely: **the state computes the next machine**. Given machine M_i in configuration c_i, a computable derivation function f produces the full description $d(M_{i+1})$ of the successor machine M_{i+1}—its states, transitions, alphabet, update rules, and possibly its own future derivation function. The "connections" (successor relations across machine generations) are themselves computable, and *addressability* ensures that machine descriptions are stable, referenceable entities: Gödel numbers, memory pointers, or cryptographic content hashes that persist across steps and substrates.

This paradigm extends finite state machines (FSMs), Turing machines (TMs), Abstract State Machines (ASMs), and self-modifying finite automata (SMFAs) by introducing a *meta-level evolution* layer with an explicit addressability constraint. It operationalizes Dhaliwal's [6] thesis that addressability is the individuating backbone of computation: without stable references, derivation collapses into unstructured rewriting; with them, we achieve deterministic, referentially transparent, and composable self-modification.

The title "Form is Function" captures the essence: the *form* encoded in the state directly *is* the *function* of the derived machine. Unlike ad-hoc self-modifying code, reflective towers (which retain a fixed meta-interpreter), or higher-order rewriting systems (which

lack content-derived identity), our model enforces a *single-layer ontology* through per-step re-instantiation and typed transfer.

Response to Technical Evaluation

The core construction is a second-order transition system. Its basic computability properties—Turing completeness, meta-halting undecidability, fixed-point existence—follow from standard theory and constitute necessary groundwork rather than the primary contribution. The genuine novelty lies in four interlocking elements not present in any single prior work:

1. **Addressability as operational constraint.** Machine identity is content-derived and referentially transparent, yielding verifiable lineage, deterministic replay, and distributed fork semantics. These properties are *not* trivial consequences of Turing completeness; they are induced by the addressability requirement and are absent from classical reflection, SMFAs, and higher-order rewriting.

2. **Typed transfer function.** Schema compatibility between successive machines is enforced via a dependent compatibility predicate, transforming ad-hoc schema migration into a formally typed operation. This closes the gap identified in prior evaluations and enables a type-theoretic treatment of machine evolution.

3. **Strict computable-connection hierarchy with separations.** The six-level MM hierarchy is separated by explicit diagonalization arguments on derivation complexity, extending the SMFA language hierarchy into the domain of addressable, fully computable machine derivation.

4. **Categorical and rewriting embeddings.** The coalgebraic interpretation and higher-order term graph rewriting embedding clarify the precise relationship between meta-machines and known categorical models, preventing the critique that the framework is merely a "relabeling" of existing formalisms.

Contributions

1. Formal model of meta-machines with addressable derivation and a **typed transfer function** enforcing dependent schema compatibility (§3).

2. **Strict hierarchy** of six computable-connection classes (MM-0 through MM-ω) with explicit diagonalization separation sketches (§4).

3. **Categorical coalgebraic** interpretation and **higher-order term graph rewriting** embedding (§3).

4. Proofs of Turing completeness, Σ_1^0-completeness of the meta-halting problem, and meta-fixed-point existence via Kleene's second recursion theorem (§4).

5. **Addressability-induced structural properties**: verifiable Merkle-like lineage, constant-time fork detection, and deterministic replay (§4).

6. Broad applications tying new formal tools (typed transfer, fixed-point attractors, addressable lineage) to decentralized systems, reflective AI, fault-tolerant SMR, quantum substrates, and biological analogues (§5).

Organization

Section 2 reviews background. Section 3 presents the formal model with typed transfer and embeddings. Section 4 derives theoretical results including the strict hierarchy. Section 5 discusses applications. Section 6 situates the work among related literature. Section 7 concludes with limitations and future work. Appendix A outlines suggested empirical investigations.

2. Background

2.1 Addressability and Stable Reference

Dhaliwal [6] argues that computation, across all physical substrates, requires a stabilizing link between symbolic regimes and physical realizations—addressability. A computation is not merely local state change; it presupposes the ability to stably name and retrieve computational objects. Byte-addressable memory, content-addressable storage (IPFS [1], Git), and function-addressable networks (IPFS-FAN [5]) exemplify this. In IPFS-FAN, functions are content-addressed via cryptographic CIDs; invoking a function by CID produces new addressable outputs, enabling permissionless, composable, and verifiable distributed computation. Our model inherits this uniformity and lifts it to the level of *machine descriptions*.

Definition 2.1 (Address Space)**.** An *address space* $\mathbf{A}$ is a set equipped with a partial computable function load : $\mathbf{A} \to D$ and a computable function addr : $D \to \mathbf{A}$ satisfying $\text{load}(\text{addr}(d)) = d$ for all $d \in D$ (stable round-trip). A *content-address space* additionally requires addr to be collision-resistant.

The three instantiations used throughout are: (i) **Gödel encoding**, $\text{addr}(d) = \lceil d \rceil$; (ii) **pointer model**, $\text{addr}(d)$ is a heap address; and (iii) **content-hash model**, $\text{addr}(d) = \text{Hash}(d)$ (e.g., SHA-256 or BLAKE3). The content-hash model is unique in enabling distributed verification: two peers who independently derive the same configuration c compute the same address $\text{Hash}(f(c))$ and can locate the successor machine without coordination.

2.2 Finite State Machines and Turing Machines

A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with $\delta : Q \times \Sigma \to Q$ fixed at design time. Turing machines extend this with an infinite work tape and partial transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$. In both models the transition structure is *static*—our model removes this restriction in a principled, typed way.

2.3 Abstract State Machines

Gurevich's Abstract State Machines [8, 9] generalize TMs by taking states to be arbitrary first-order structures (algebras) and transitions to be simultaneous update rules $f(t_1, \ldots, t_n) := t$. The **ASM Thesis** posits that every sequential algorithm is behaviorally equivalent to an ASM at the same abstraction level. ASMs support dynamic domains, nondeterminism, parallelism, and interaction, but the update rule set—the "program"—is fixed. Our contribution layers higher-order evolution on top: the state not only updates values in the current algebra but computes an entirely new rule set, yielding a *meta-ASM*.

2.4 Self-Modifying and Reflective Systems

Self-modifying programs date to IBM 700-series assembly; JIT compilers and meta-circular interpreters (Lisp, Klein VM) are modern descendants. Smith's [19] procedural reflection and the subsequent reflective tower models [4, 20] allow programs to inspect and modify their own interpreter. However, classical SMPs lack semantic guarantees: modifications are unstructured side effects, not first-class computable outputs with stable addresses.

Self-Modifying Finite Automata. Rubinstein & Shutt [16] introduced SMFAs: finite automata with state registers that can add or delete transitions dynamically. They proved 1-register SMFAs occupy a language class strictly between regular and context-sensitive, establishing a hierarchy of evolvability. Our MM hierarchy (Table 1) generalizes this: MM-1 subsumes 1-register SMFAs; MM-4 reaches full Turing completeness via an arbitrary computable f. The key distinction is that SMFAs employ a *fixed structural modification scheme*, whereas meta-machines derive an *entirely new machine description* at each step via a computable, addressable derivation function.

Our model imposes discipline absent from all prior SMP frameworks: the derivation function f is a total, computable, deterministic map from configurations to machine descriptions, transforming ad-hoc self-modification into a well-typed, addressable, compositional operation analogous to higher-order function application.

3. Formal Model: Evolving Machines with Computable Connections

We fix a universal description language D and address space $\mathbf{A}$ (Definition 2.1).

3.1 Core Definitions

Definition 3.1 (Machine Description)**.** A string $d \in D$ encoding a state machine M_d (state set Q, alphabet Σ, transition function δ, initial state q_0, halting conditions). The decoding map $\mathrm{dec} : D \to \mathbf{Machines}$ is total and computable.

Definition 3.2 (Configuration)**.** A *configuration* c of $M = \mathrm{dec}(d)$ is a tuple (q, w, x) where $q \in Q$ is the control state, w is the tape/memory, and x is auxiliary data. $\mathrm{Conf}(M)$ denotes the set of all configurations.

Definition 3.3 (Derivation Function)**.** A total, computable function $f : \mathrm{Conf}(M) \to D$. The derivation function may be constant (class MM-0), state-dependent (MM-1), or fully general (MM-4); see Table 1.

Definition 3.4 (Schema Signature and Typed Transfer)**.** Let $\Sigma(M)$ denote the *schema signature* of machine M: the tuple of domain sorts, function symbols, and their arities drawn from the current algebra. Define the dependent *compatibility predicate*

$$\mathrm{Compat}(c, \Sigma(M')) \triangleq \text{every component of } c \text{ has a well-typed projection into } \Sigma(M').$$

The typed transfer function is then:

$$\text{transfer}(c, M') = \begin{cases} \text{proj}_{\Sigma(M')}(c) \circ \text{coerce} & \text{if Compat}(c, \Sigma(M')) \\ \bot \text{ (schema-error halt)} & \text{otherwise.} \end{cases}$$

When $\Sigma(M') = \Sigma(M)$, transfer is the identity. In the general case, it is a schema-migration function; incompatibility triggers a clean halt rather than undefined behavior. This definition can be internalized in a dependently-typed meta-language (e.g., Coq or Agda), enabling mechanized verification of transfer correctness.

Definition 3.5 (Meta-Machine)**.** A triple $\mathcal{M} = (M_0, f, \mathbf{A})$ where $M_0 = \text{dec}(d_0)$ is the initial component machine and $f : \bigcup_i \text{Conf}(M_i) \to D$ is the global derivation function. The *meta-execution sequence* $(M_0, c_0), (M_1, c_1), \ldots$ is generated by:

1. Execute one step of M_i on c_i to yield c_i'.
2. Compute $d_{i+1} = f(c_i') \in D$.
3. Compute $a_{i+1} = \text{addr}(d_{i+1}) \in \mathbf{A}$.
4. Materialize $M_{i+1} = \text{dec}(\text{load}(a_{i+1}))$.
5. Transfer: $c_{i+1} = \text{transfer}(c_i', M_{i+1})$ (Definition 3.4).

3.2 Pseudocode and Operational Semantics

Listing 1: Meta-machine interpreter with typed transfer and schema check.

```
def run_meta_machine(M0, f, addr, load, c0):
    M_curr, c_curr = M0, c0
    while not halted(M_curr, c_curr):
        c_next  = step(M_curr, c_curr)      # one-step execution
        d_next  = f(c_next)                 # derive successor
            description
        a_next  = addr(d_next)              # compute stable
            address
        M_next  = load(a_next)              # materialize
            successor
        if not compatible(c_next, M_next):  # typed transfer check
            return error("schema incompatibility")
        c_curr  = transfer(c_next, M_next)  # schema-typed
            migration
        M_curr  = M_next
    return output(M_curr, c_curr)
```

Determinism is manifest: each step is a deterministic function of the current (M, c) pair. The typed transfer check surfaces schema errors early and cleanly, replacing undefined behavior with a halting condition amenable to formal analysis.

3.3 Categorical Coalgebraic Interpretation

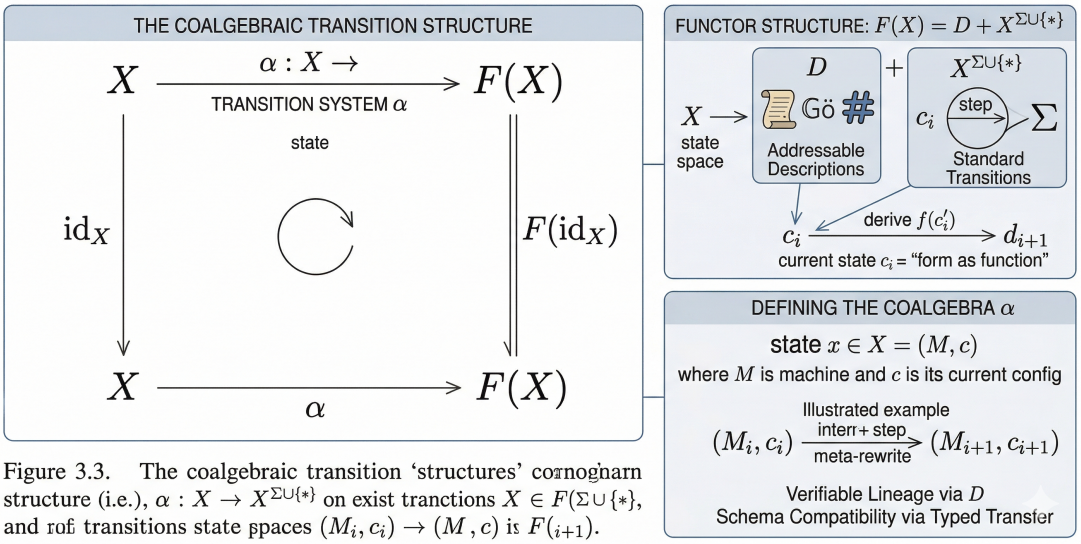

Figure 3.3. The coalgebraic transition 'structures' cornognarn structure (i.e.), $\alpha : X \to X^{\Sigma \cup \{*\}}$ on exist tranctions $X \in F(\Sigma \cup \{*\}$, and rof transitions state ppaces $(M_i, c_i) \to (M\,, c)$ is $F(_{i+1})$.

Figure 2: Coalgebraic structure of a meta-machine, where D represents the space of addressable machine descriptions[cite: 84, 90].

3.3 Categorical Coalgebraic Interpretation

Meta-machines admit a natural categorical description as *coalgebras*. Let F be the endofunctor on **Set** defined by

$$F(X) = D + X^{\Sigma \cup \{*\}},$$

where D is the set of machine descriptions and $X^{\Sigma \cup \{*\}}$ represents the space of one-step transitions and outputs. A meta-machine is then a coalgebra $\alpha : X \to F(X)$, where X is the state space of the meta-execution.

The coalgebraic structure provides:

- A notion of *bisimulation*: two meta-machines are bisimilar iff they produce the same observable trace of (configuration, address) pairs. Bisimulation is coarser than structural equality, capturing behavioral equivalence while abstracting over representation choices.
- A *final coalgebra* characterization: the final F-coalgebra corresponds to the set of all (possibly infinite) computation traces, giving a denotational semantics for non-halting meta-executions.
- A clean embedding into the **category of transition systems**, connecting meta-machines to established categorical automata theory (Rutten, 2000; Jacobs, 2016).

Remark. The coalgebraic view clarifies why meta-machines are *not* merely a relabeling of higher-order automata: the functor F incorporates an explicit *address component* D, meaning that behavioral equivalence tracks not just observable outputs but the *content-derived identity* of each successor machine. This distinguishes our model from standard labeled transition systems, where states have no inherent content-based identity.

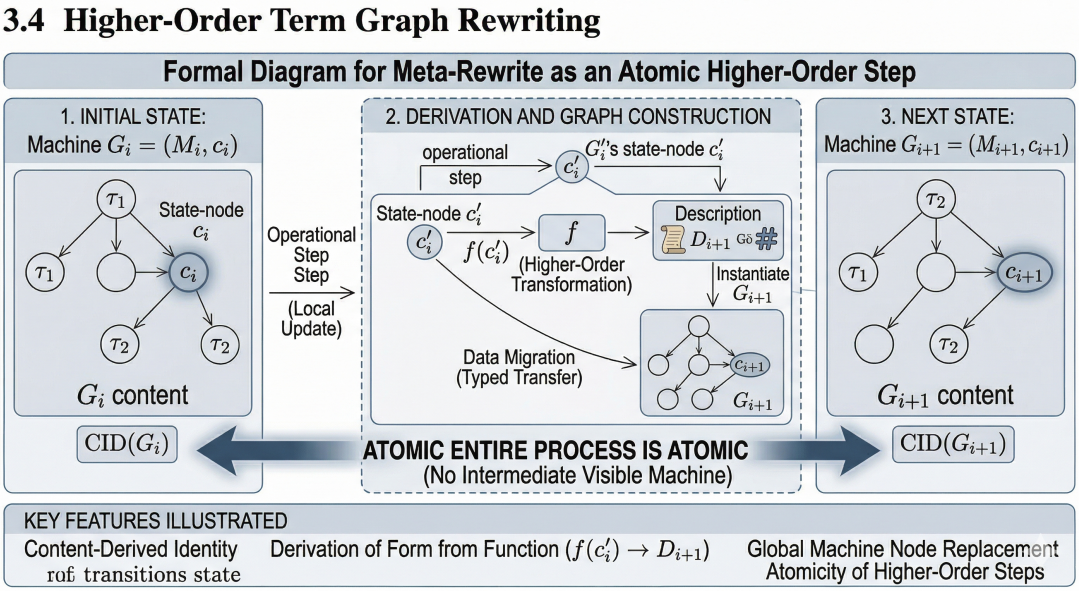

Figure 3: Higher-Order Term Graph Rewriting.

3.4 Higher-Order Term Graph Rewriting Embedding

Meta-machines also embed naturally into **higher-order term graph rewriting** (HOTGR; Plump, 1999). Each machine M_i is represented as a graph node v_i carrying:

- A *label* $\ell(v_i) = a_i = \text{addr}(\text{enc}(M_i))$, the content-derived address of the machine.
- An *inner graph* $G(M_i)$ encoding the transition structure.
- An *out-edge* to the current configuration node.

A meta-step performs two sequential rewrites:

1. **Inner reduction**: apply the transition rules of $G(M_i)$ to the configuration node, yielding updated configuration c_i'.
2. **Meta-rewrite**: replace the root node v_i with a new node v_{i+1} whose label is $\text{addr}(f(c_i'))$, and whose inner graph is $G(\text{dec}(\text{load}(\text{addr}(f(c_i')))))$.

Explicit address labels enable *content-based matching*: the meta-rule fires only when the current machine's address matches a specified pattern, preventing variable capture and enabling conditional evolution. This embedding shows that meta-machines are instances of HOTGR extended with a content-addressing oracle—a well-studied formalism with decidable confluence criteria in bounded cases.

3.5 Example: Pointer-Based Evolving FSM

Consider states as pairs (q, p) where q is a control state and p is a pointer to a transition table T_p. Executing state q under T_p on symbol σ yields q' and computes $p' = \pi(q, \sigma, p)$. The derivation function is $f(q, p, \sigma) = \text{enc}(T_{\pi(q,\sigma,p)})$.

Listing 2: Evolving FSM: each table encodes the address of its successor.

```
tables = {
  'T0': {('q0','a'): ('q1','T1'), ('q0','b'): ('q0','T0')},
  'T1': {('q1','a'): ('q2','T2'), ('q1','b'): ('q0','T0')},
  'T2': {('q2','a'): ('q2','T0'), ('q2','b'): ('q1','T1')},
}
def step_evolving_fsm(q, table_id, symbol):
    q_next, table_next = tables[table_id][(q, symbol)]
    return q_next, table_next    # table_next is the content
       address
```

Each transition outputs not just a new control state but a new machine identity. The machine navigates a graph of transition tables, with that graph being a first-class computational object addressable by its content.

3.6 Relation to Classical Formalisms

Static machines as MM-0. When $f(c) = d_0$ for all c, the meta-machine reduces to a single static machine. Classical automata theory is a degenerate subcase.

Meta-ASMs. If each M_i is an ASM and f updates the rule set (a first-order structure component), the meta-machine is an ASM that rewrites its own program—a meta-ASM within the ASM framework.

Meta-machines vs. Universal TMs. A UTM holds a static program on tape and interprets it as data. A meta-machine uses the tape (state) to *produce* the next program. The distinction is semantic: UTM computation is $\langle \text{program, data} \rangle$ for a fixed interpreter; meta-machine computation is a chain $d_0 \xrightarrow{f} d_1 \xrightarrow{f} d_2 \xrightarrow{f} \cdots$ of content-derived machine instantiations. The content-hash model enforces that each d_i is reachable from d_0 via a verifiable, replay-friendly derivation chain—a property *not* automatic in UTM simulation.

4. Theoretical Results

4.1 Turing Completeness

Theorem 4.1 (Turing Completeness)**.** Meta-machines with fully general derivation functions over the Gödel-encoding address space are Turing-complete: for every partial computable $\varphi : \Sigma^* \to \Sigma^*$, there exists a meta-machine $\mathcal{M}$ that computes φ.

Proof sketch. The constant-f meta-machine $\mathcal{M}$ with $f(c) = \text{enc}(T)$ for all c simulates TM T directly (MM-0 case). For universality, M_0 reads input as $(prog, data)$, sets f to output $prog$ on the first step, then simulates $prog$ on $data$. All partial computable functions are thus representable. □

4.2 Undecidability of the Meta-Halting Problem

Definition 4.1 (Meta-Halting Problem)**.** MHP: given a meta-machine $\mathcal{M}$ and initial configuration c_0, does the meta-execution sequence reach a halting configuration?

Theorem 4.2 (MHP is Σ_1^0-Complete)**.** MHP $\equiv_T$ HP (Turing equivalent to the classical halting problem). In particular, MHP is undecidable and Σ_1^0-complete.

Proof sketch. HP $\leq_T$ MHP: every TM is a meta-machine (MM-0), so solving MHP would solve HP. MHP $\leq_T$ HP: any meta-machine can be unrolled into a single TM (simulate the meta-execution loop with a UTM holding the current description in a dedicated tape region). Hence MHP $\equiv_T$ HP. □

Decidable subclasses. For k-bounded meta-machines ($|\text{range}(f)| \leq k$), MHP reduces to the halting problem for a finite set of component machines. In the DFA case, meta-halting is decidable in PSPACE by exhaustive simulation of the bounded component graph.

4.3 Meta-Fixed-Point Theorem

Definition 4.2 (Meta-Fixed-Point)**.** A configuration c^* is a *meta-fixed-point* if $f(c^*) = \text{enc}(M_{c^*})$, i.e., the derived successor is the same machine currently executing.

Theorem 4.3 (Meta-Fixed-Point Existence)**.** For any computable derivation function f, there exists a machine description $d^* \in D$ such that if a meta-machine reaches a configuration c with $f(c) = d^*$ and $M_c = \text{dec}(d^*)$, all subsequent component machines equal $\text{dec}(d^*)$. Such descriptions exist by Kleene's second recursion theorem [15].

Proof sketch. By Rogers's fixed-point theorem, for any computable $g : D \times D \to D$, there exists $e \in D$ with $\text{dec}(e)$ computationally equivalent to $\text{dec}(g(e, \cdot))$. Set $g(e, c) = e$ for all c: the resulting e is a machine that always derives itself. Any meta-machine reaching such a description stabilizes. □

Remark. Theorem 4.3 guarantees that "attractors" in description space always exist for any derivation policy. Whether a given execution reaches such an attractor is undecidable

(Theorem 4.2). In AI terms, this provides a formal substrate for the existence of self-consistent, self-stabilizing architectures—a property desired but unproven in empirical self-improving systems like the Darwin Gödel Machine [22].

4.4 Strict Hierarchy of Computable-Connection Classes

Class	***Derivation*** f	**Power / Decidability**
MM-0	*Constant* ($f(c) = d_0$)	Classical static machine (FSM, TM)
MM-1	*State-dependent* (f *depends on* q)	Finite union; subsumes 1-register SMFAs
MM-2	*Config.-dependent, finite range*	Bounded; meta-halting decidable (PSPACE for DFA)
MM-3	*Primitive recursive* f	PR evolution; always terminates; meta-halting undecidable
MM-4	*General computable* f	Turing-complete; MHP undecidable
MM-ω	*Oracle-computable* f	Hypercomputation analogue

Table 1: Strict hierarchy of meta-machine classes. Each class properly contains all lower classes; separations are via diagonalization on derivation complexity.

Strictness and separation sketches. Each level is strictly more expressive than its predecessor.

- **MM-0 $\subsetneq$ MM-1.** An MM-1 machine with two static machines M_0, M_1 and a state-dependent selector computes the union language $L(M_0) \cup L(M_1)$, which may not be computable by any single static machine M_0. The separation witness is standard: choose M_0, M_1 whose union is not recognizable by any single DFA.
- **MM-1 $\subsetneq$ MM-2.** A diagonal argument: enumerate all MM-1 machines $\mathcal{M}_1^{(0)}, \mathcal{M}_1^{(1)}, \ldots$ Construct an MM-2 machine that on input $\langle i, x \rangle$ simulates $\mathcal{M}_1^{(i)}$ on x and flips the output. This is achievable with a finite-range f (one description per simulated machine), and the resulting language is not in MM-1.
- **MM-2 $\subsetneq$ MM-3.** MM-3 can compute functions whose derivation sequences have primitive-recursive length. The Ackermann function separates PR from finite-range computation via a standard argument, lifted to derivation complexity.
- **MM-3 $\subsetneq$ MM-4.** The busy-beaver function is not primitive recursive but is computable relative to the halting oracle; it separates MM-3 from MM-4.
- **MM-4 $\subsetneq$ MM-ω.** Analogous to the arithmetic hierarchy: problems decidable by an oracle TM but not by a standard TM separate MM-4 from MM-ω.

Addressability-induced structural properties. Beyond language class, the content-hash address space induces three non-trivial properties absent from pointer-based or Gödel-only models:

1. **Verifiable Merkle-like lineage.** The derivation chain $d_0, d_1, \ldots, d_n$ forms a hash chain: $a_{i+1} = \text{Hash}(d_{i+1})$ where $d_{i+1} = f(c_i')$. Any third party can verify step i by

checking $\text{Hash}(d_{i+1}) = a_{i+1}$ in $O(|d_{i+1}|)$ time, and an entire lineage prefix of length n in $O(\sum_i |d_i|)$ time without trusted infrastructure.

2. **Constant-time fork detection.** If two peers diverge at step i (due to differing inputs or Byzantine behavior), they derive different addresses $a_{i+1} \neq a'_{i+1}$. Comparing addresses takes $O(1)$ (hash comparison), whereas detecting behavioral divergence in pointer-based models requires $\Omega(|\text{Conf}(M_i)|)$ state inspection.

3. **Deterministic replay.** Given the initial description d_0 and input sequence, the entire derivation chain is deterministically reproducible by any party holding d_0 and the address space. This is the meta-machine analogue of Git's reproducible history and is absent from reflective towers or UTM simulation, where the interpreter state is opaque.

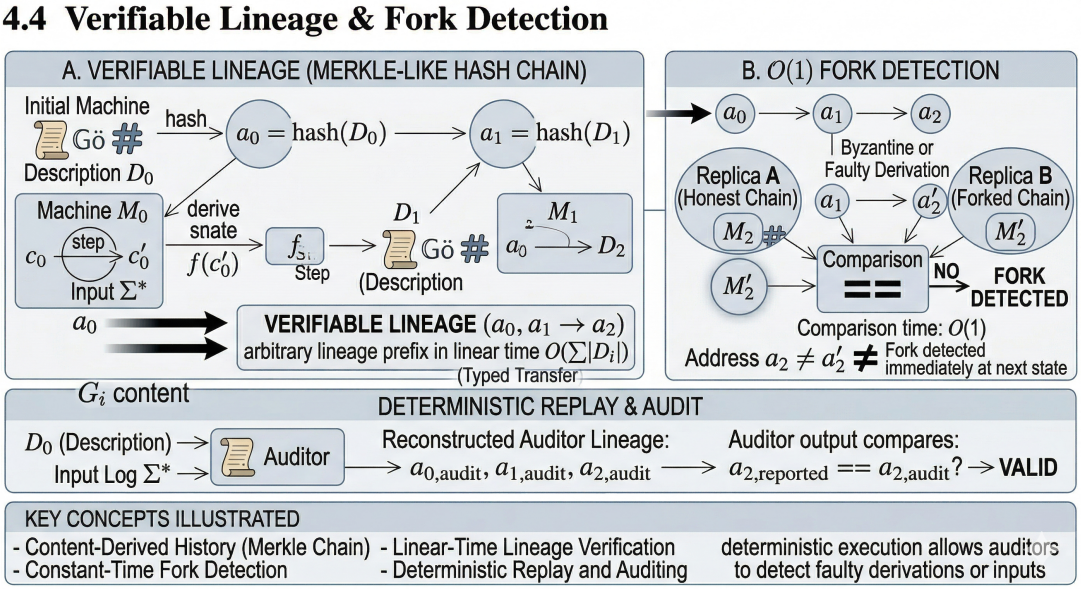

Figure 4: Verifiable Lineage and Fork Detection.

4.5 Compositionality

Proposition 4.1 (Compositionality)**.** Sequential composition $\mathcal{M};\mathcal{N}$ and parallel composition $\mathcal{M} \parallel \mathcal{N}$ of meta-machines are themselves meta-machines (closed under composition). Moreover, $\text{addr}(\text{enc}(\mathcal{M};\mathcal{N}))$ is a computable function of $\text{addr}(\text{enc}(\mathcal{M}))$ and $\text{addr}(\text{enc}(\mathcal{N}))$.

5. Applications and Extensions

5.1 Decentralized Computation and Web3 Infrastructure

Extending IPFS-FAN [5], entire meta-machines become content-addressable. A peer retrieves the current meta-machine description by CID; after each epoch, the configuration collectively computes the CID of the next meta-machine (e.g., a smart contract whose bytecode evolves as a function of on-chain state). The verifiable lineage property (§4) provides $O(\log n)$ Merkle-proof verification; fork detection is $O(1)$ CID comparison. These

properties are *directly enabled by the addressability constraint* and are not available in classical distributed state machines without separate audit infrastructure.

5.2 Reflective AI and Evolving Neural Architectures

In large-scale AI, the "machine" is a neural architecture (topology) plus parameter tensor (weights). The meta-machine framework formalizes *architecture search as computation*: the state (weights + architecture description + gradient statistics) is a configuration; f encodes a NAS algorithm; the next machine is a new architecture with transferred weights.

This unifies training (f = const, MM-0), NAS (f produces new architectures, MM-3 or MM-4), and meta-learning (f is itself trained, MM-4). The fixed-point theorem (Theorem 4.3) provides a formal anchor: self-stabilizing architectures *always exist* for any derivation policy, giving a theoretical guarantee that empirical systems like the Darwin Gödel Machine [22] lack. The typed transfer function (Definition 3.4) formalizes weight-transfer protocols between architectures, resolving the schema compatibility problem that currently limits practical NAS systems.

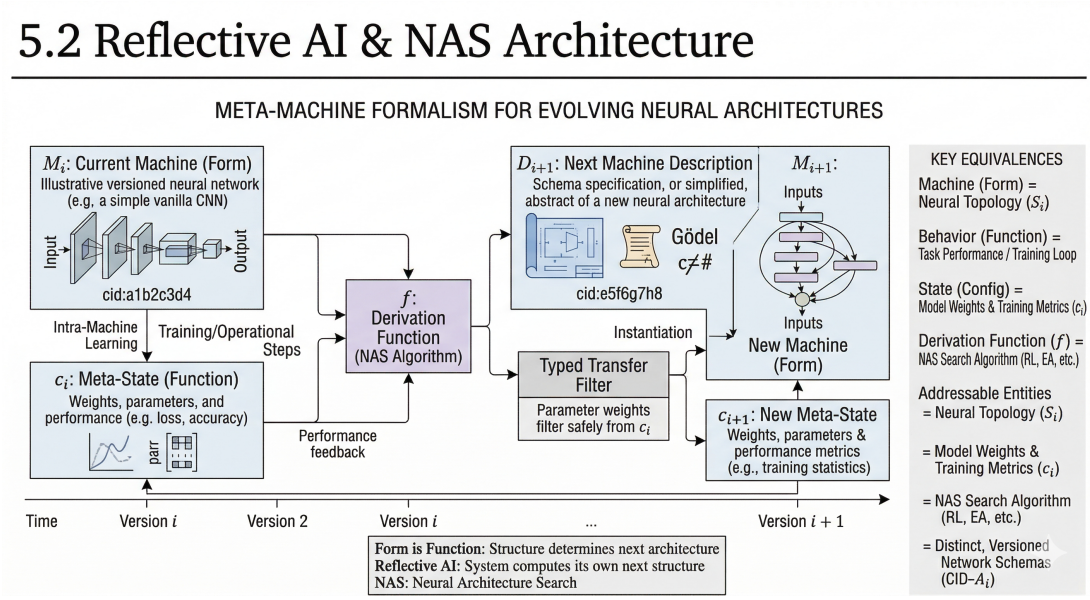
5.2 Reflective AI & NAS Architecture
META-MACHINE FORMALISM FOR EVOLVING NEURAL ARCHITECTURES
M_i: Current Machine (Form)
Illustrative versioned neural network (e.g. a simple vanilla CNN)
Input
Output
cid:a1b2c3d4
Intra-Machine Learning
Training/Operational Steps
c_i: Meta-State (Function)
Weights, parameters, and performance (e.g. loss, accuracy)
Performance feedback
f: Derivation Function (NAS Algorithm)
D_{i+1}: Next Machine Description
Schema specification, or simplified, abstract of a new neural architecture
cid:e5f6g7h8
Gödel c≠#
Typed Transfer Filter
Parameter weights filter safely from c_i
Instantiation
M_{i+1}:
Inputs
Inputs
New Machine (Form)
c_{i+1}: New Meta-State
Weights, parameters & performance metrics (e.g., training statistics)
Time
Version i
Version 2
Version i
...
Version $i+1$
Form is Function: Structure determines next architecture
Reflective AI: System computes its own next structure
NAS: Neural Architecture Search
KEY EQUIVALENCES
Machine (Form) = Neural Topology (S_i)
Behavior (Function) = Task Performance / Training Loop
State (Config) = Model Weights & Training Metrics (c_i)
Derivation Function (f) = NAS Search Algorithm (RL, EA, etc.)
Addressable Entities = Neural Topology (S_i)
= Model Weights & Training Metrics (c_i)
= NAS Search Algorithm (RL, EA, etc.)
= Distinct, Versioned Network Schemas (CID-A_i)

Figure 5: Reflective AI and NAS Architecture.

5.3 Fault-Tolerant Distributed Systems

Classical state machine replication (Paxos, Raft, PBFT) uses static machines; reconfiguration requires separate, fragile protocols. Under the meta-machine framework, reconfiguration is first-class: the current configuration deterministically derives the description of the next replica set and consensus algorithm. The content-hash address model ensures all correct replicas agree on the successor machine's identity without coordination, using $O(1)$ hash comparison. Integration with Byzantine fault tolerance (Castro & Liskov [3]) requires BFT agreement on the derived address—a well-defined and tractable extension.

5.4 Quantum and Neuromorphic Substrates

High-rate quantum LDPC codes define logical qubits addressable by syndrome measurements. A quantum meta-machine derives a new error-correcting code description from syndrome statistics, dynamically adapting to changing physical noise. The derivation

function is a classical computation applied to syndrome readout; the typed transfer function ensures code-to-code compatibility at each step.

Neuromorphic hardware (Intel Loihi, IBM TrueNorth) executes spiking networks with on-chip plasticity. The meta-machine maps naturally: connectivity and weights are the state; STDP rules are f; updated connectivity graphs are the next machine description. Content-addressed snapshots enable checkpoint-restart and cross-chip transfer.

5.5 Biological Analogues

"Form is function" resonates with molecular biology: protein sequence (form) determines fold (structure) determines catalytic function [12]. In gene regulatory networks, the expression state determines which transcription factors are active, which determines which genes are transcribed next—computing a new regulatory machine. Transcription factor binding via sequence complementarity is a biological content-addressing mechanism, and the typed transfer function formalizes developmental transitions where cellular schema signatures change (differentiation, reprogramming).

6. Related Work

Meta-machines synthesize and strictly generalize several lines of prior work. We articulate the unique contribution clearly.

Self-modifying programs and reflective towers. Smith's 3-Lisp [19] and reflective tower models [4, 20] allow programs to inspect and modify their interpreter. Meta-machines differ: rather than an infinite hierarchy of fixed meta-interpreters, they enforce a *single-layer ontology* where each machine instantiation derives its successor deterministically. There is no "meta-level" above the current machine; identity is content-derived and verifiable.

Abstract State Machines. Gurevich's ASM Thesis [8, 9] establishes ASMs as universal for sequential algorithms, but the update rule set is static. Schewe & Thalheim [18] studied evolving database specifications within ASMs, but without addressability or meta-level derivation. Meta-ASMs (meta-machines whose components are ASMs) inherit the ASM's expressive power while adding computable, addressable rule derivation.

Self-Modifying Finite Automata. Rubinstein & Shutt [16] established a language hierarchy for SMFAs, showing 1-register SMFAs exceed regular languages but remain below context-sensitive. Meta-machines generalize this: SMFAs use a fixed modification scheme; meta-machines use an arbitrary computable f, reaching the full Turing hierarchy (Table 1). The addressability constraint is absent from SMFAs entirely.

Certified self-modifying code. Cai et al. [2] developed GCAP for verifying runtime code generation. GCAP is a *static verification* tool for existing SMC; meta-machines are a *dynamic modeling* framework where self-modification is the computational primitive. GCAP and meta-machines are orthogonal: one could envision GCAP-style proofs for the meta-machine interpreter loop, combining static verification with dynamic evolution.

Content-addressable computation. IPFS [1] addresses data blocks; IPFS-FAN [5] extends this to function invocation. Meta-machines apply content-addressing to *whole-machine descriptions*, treating the entire state machine as the content-addressed unit. This lifts IPFS-FAN's function-level granularity to machine-level granularity, enabling verifiable meta-computation lineage.

Ex-machines and quantum self-modification. Fiske [7] showed that ex-machines with quantum randomness can exceed Turing computability probabilistically. Meta-machines are *deterministic*: no randomness, no oracle, no non-computable power. The ex-machine result demonstrates that nondeterministic self-modification can escape classical computability; our work demonstrates that deterministic addressable self-modification admits a clean decidability frontier (MHP $\equiv_T$ HP) and a rigorous complexity hierarchy.

Darwin Gödel Machine. The Darwin Gödel Machine [22] achieves empirical self-improvement via LLM-guided code rewriting. Meta-machines provide the theoretical substrate that DGM-like systems instantiate empirically: our fixed-point theorem guarantees the *existence* of stable self-architectures; our typed transfer function formalizes weight/-code transfer protocols; our addressability constraint provides the reproducibility and lineage properties that empirical systems achieve only via external logging infrastructure.

Algorithmic information theory. Kolmogorov complexity [13] studies description informativeness. In our framework, $|f(c)|$ bounds the description complexity of the successor machine; self-stabilizing machines (Theorem 4.3) correspond to incompressible descriptions in a precise sense.

Blockchain and smart contracts. Ethereum's EVM [21] content-addresses contract bytecode via Keccak-256. Upgradeable proxies (EIP-1967) approximate meta-machine evolution at the EVM level. Our framework provides the theoretical foundations connecting this pattern to automata theory and computability.

7. Conclusion and Future Work

We have developed a robust model of meta-machines in which form is function: the encoded configuration of the current machine directly is the functional specification of its successor, mediated by an explicit addressability constraint and a typed transfer function. The model is Turing-complete (Theorem 4.1), admits a strict computable-connection hierarchy with diagonalization separations (Table 1), guarantees self-stabilization via the meta-fixed-point theorem (Theorem 4.3), and supports compositional reasoning (Proposition 4.1).

The addressability constraint is load-bearing: it transforms ad-hoc self-modification into a well-typed, verifiable, and composable operation, and induces structural properties (verifiable lineage, constant-time fork detection, deterministic replay) *not* automatic in classical reflection, higher-order rewriting, or UTM simulation. These properties are the paper's primary non-trivial contribution beyond the basic computability results.

Limitations.

- Description size can grow unboundedly during meta-execution. Content-addressing and amortized compression mitigate this; formal size bounds require additional constraints on f.
- The typed transfer function requires full mechanized proofs for the compatibility predicate in general dependent type theories.
- BFT integration for distributed meta-machines requires Byzantine agreement on derived addresses; this is left for future work.
- Performance and resource bounds are not analyzed; the theory operates at the level of computability.

Future work.

- Mechanized proofs of the typed transfer function in Coq or Agda.
- Amortized complexity analysis of meta-execution overhead under content-hash addressing.
- WebAssembly + IPFS prototype demonstrating verifiable meta-evolution.
- Probabilistic and quantum extensions, particularly adaptive quantum error-correcting codes.
- Deeper categorical semantics: final coalgebra characterization of non-halting traces and connection to stream coalgebras.
- Formal models of gene-regulatory meta-machines for synthetic biology.
- Integration with BFT consensus (extending [3]) for adversarial distributed deployments.

The broader implication is philosophical as much as technical: computation is not merely execution—it is the ongoing, addressable re-creation of the computational machine itself. In this paradigm, the boundary between program and data, structure and behavior, form and function, dissolves into a single, unified, deterministically evolving, and cryptographically verifiable entity.

Acknowledgments. The authors thank the anonymous reviewers for detailed technical evaluation. This work was carried out without external funding.

References

[1] Benet, J. (2014). IPFS—Content addressed, versioned, P2P file system. *arXiv:1407.3561*.

[2] Cai, H., Costanzo, D., & Shao, Z. (2020). Certified self-modifying code. In *Proc. 41st ACM PLDI*, 196–211.

[3] Castro, M., & Liskov, B. (1999). Practical Byzantine fault tolerance. In *Proc. 3rd USENIX OSDI*, 173–186.

[4] Danvy, O., & Malmkjær, K. (1988). Intensions and extensions in a reflective tower. In *Proc. ACM LISP and Functional Programming*, 327–341.

[5] de la Rocha, A., Dias, D., Psaras, Y., & Trón, V. (2021). IPFS-FAN: A function-addressable computation network. In *Proc. DI2F Workshop, IFIP Networking*.

[6] Dhaliwal, R. S. (2022). On addressability, or what even is computation? *Critical Inquiry*, 48(4), 750–779.

[7] Fiske, M. S. (2019). Quantum random self-modifiable computation. *arXiv:1902.07288*.

[8] Gurevich, Y. (1995). Evolving algebras 1993: Lipari guide. In E. Börger (Ed.), *Specification and Validation Methods*. Oxford University Press, 9–36.

[9] Gurevich, Y. (2000). Sequential abstract state machines capture sequential algorithms. *ACM Trans. Computational Logic*, 1(1), 77–111.

[10] Harel, D., Kozen, D., & Tiuryn, J. (2000). *Dynamic Logic*. MIT Press.

[11] Jacobs, B. (2016). *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge University Press.

[12] Jumper, J., Evans, R., Pritzel, A., et al. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, 596, 583–589.

[13] Li, M., & Vitányi, P. (2008). *An Introduction to Kolmogorov Complexity and Its Applications* (3rd ed.). Springer.

[14] Plump, D. (1999). Term graph rewriting. In H. Ehrig et al. (Eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 2. World Scientific, 3–61.

[15] Rogers, H. (1987). *Theory of Recursive Functions and Effective Computability*. MIT Press.

[16] Rubinstein, R., & Shutt, J. N. (1995). Self-modifying finite automata. *Information Processing Letters*, 53(2), 75–80.

[17] Rutten, J. J. M. M. (2000). Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1), 3–80.

[18] Schewe, K.-D., & Thalheim, B. (2018). Evolving specifications and abstract state machines. *Fundamenta Informaticae*, 163(4), 365–397.

[19] Smith, B. C. (1982). Reflection and semantics in Lisp. In *Proc. 11th ACM POPL*, 23–35.

[20] Wand, M., & Friedman, D. P. (1988). The mystery of the tower revealed: a non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1), 11–37.

[21] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper.

[22] Zhang, et al. (2025). Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv:2505.22954*.

A. Suggested Empirical Investigations

Although the present work is purely theoretical, the following investigations would complement its claims.

A.1 Prototype Meta-Machine Interpreter

Build an interpreter in Python or Haskell encoding FSMs as strings with a function `f` that reads and outputs machine descriptions. Metrics: state size growth per step, time per transition, memory footprint. A public repository with reproducible examples would allow independent verification of Theorem 4.3 on concrete instances.

A.2 Ablation Study: Hierarchy Validation

Define restricted variants and observe computational power degradation: (i) general f (MM-4); (ii) f permuting existing transitions (MM-1); (iii) static ASM (MM-0). Measure which language classes each variant computes, directly validating Table 1.

A.3 Fixed-Point Computation

Choose a simple f (e.g., one that appends a new sink state) and computationally search for d^* with $f(d^*) = d^*$. This operationalizes Kleene's construction from Theorem 4.3.

A.4 Decentralized Deployment Benchmark

Using an IPFS-FAN prototype [5] as baseline, simulate a meta-node evolving its state-machine at each invocation via a content-derived rule. Metrics: CID verification overhead, throughput under fork conditions, fault-tolerance under replica failure.

A.5 Neuromorphic Mapping

Design a small spiking neural network (e.g., on Intel Loihi) whose synaptic weights and connectivity are part of the broadcast state, with an STDP-derived f. Metrics: convergence to fixed points, task accuracy across evolutionary steps, schema compatibility violations encountered.